Module2: Inheritance, Packages and Interfaces:

1. Create an Array of Objects

2. Understand the Principle of OOP - Inheritance

- Problem Solving Assignment
- 1. Create an array of products with data id, name, price and sale. Display the objects in ascending order of sales.
- 2. Describe the various forms of inheritance in Java.

1. Array of Objects

```
/* Array of Objects
datatype arr_name[] = new
type[size];
Student s[ ] = new Student[10];
datatype[] arr_name = new
type[size];
Student[] s = new Student[10]; */
import java.util.*;
class Student
{
int id ;
String name;
double avg;
Student(int id)
{ this.id = id; }
Student(int id , String name )
{
this(id);
this.name= name;
}
Student (int id , String name ,
double avg)
{
this(id,name);
this.avg = avg; }
void display( )
{
System.out.println(id + " , " + name
+ ", " + avg);
} }
public class Filter
{
public static void main(String
args[])
{
Scanner sc = new
```

```
for(int i=0; i<s.length ; i++)
{
System.out.println("Enter the
Student Details of -> " +(i+1));
int x = sc.nextInt();
String str = sc.next();
double y = sc.nextDouble();
s[i] = new Student(x,str,y);
}
System.out.println("Enter the
Average to display list ");
double p = sc.nextDouble();

for(int i=0; i< n ; i++)
if(s[i].avg >= p)
s[i].display();
}
}
```

2. Inheritance in Java

Inheritance in java is a mechanism in which one object acquires the properties and behaviors of another object.

The idea behind inheritance in java is that we can create new classes that are built upon existing classes.

When we inherit from an existing class, we can reuse methods & data of parent class; we can add new methods and data also.

Inheritance represents the IS-A relationship, also known as *parent-child* relationship.

Need of inheritance in java

- **For Method overriding (so runtime Polymorphism can be achieved).**
- **For Code Reusability, Extend the functionality.**
- **It allows creation of Hierarchical Classification.**

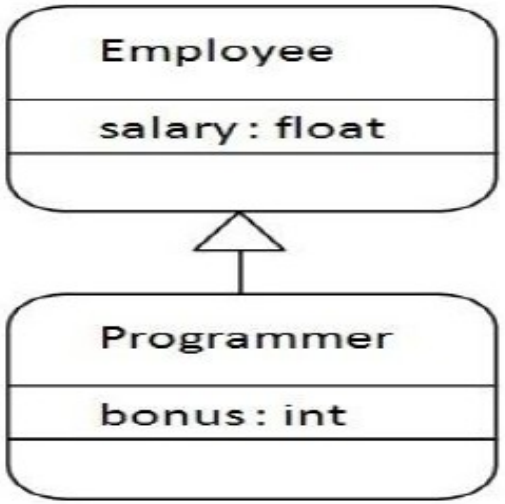
Syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
//methods and fields of sub class
}
```

The extends keyword indicates that we are making a new class that derives from an existing class.

- In the terminology of Java, a class which is inherited is called **parent or super class**.
- The class that does the inheriting is called (the new class) is called **child or subclass.**
- A subclass is a specialized version of a super class. It inherits the members defined by the super class and adds its own unique elements.
- Inheritance represents the **IS-A** relationship, also called

Java Inheritance Example



ımmer Is-A Employee

- Here **Programmer** is the subclass and **Employee** is the super class.

Relationship between two classes is Programmer IS-A Employee.

It means that Programmer is a type of Employee.

```
class Employee
{
float salary=40000;
}
class Programmer extends Employee
{
float bonus=(0.2)*salary;
}
public class InhDemo
{
public static void main(String args[])
{
Programmer p=new Programmer();
System.out.println("Programmer salary
is:"+p.salay);
System.out.println("Bonus of Programmer
is:"+p.bonus);
}
}
```

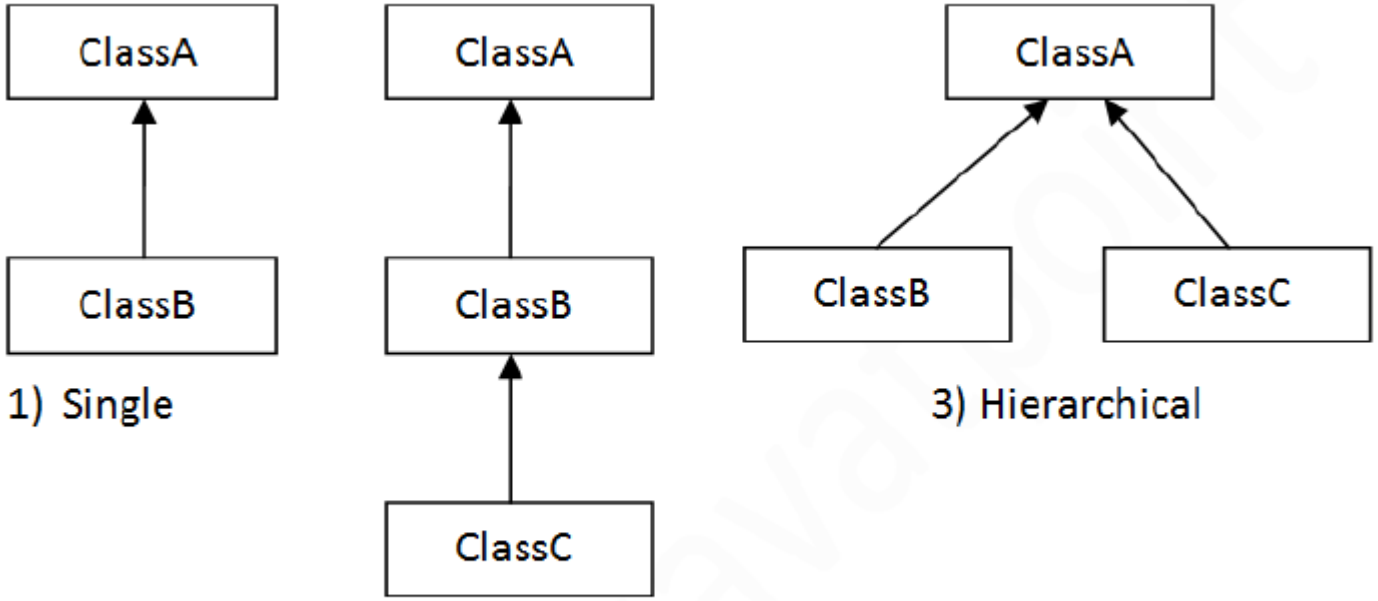
```
Programmer salary is: 40000.0
```

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

***A major advantage of inheritance is that once we created a super class that defines the attributes common to a set of objects, it can be used to create any number

Types of inheritance in java

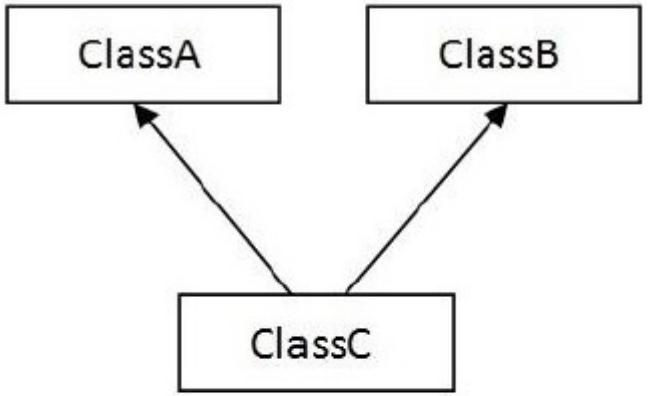
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.



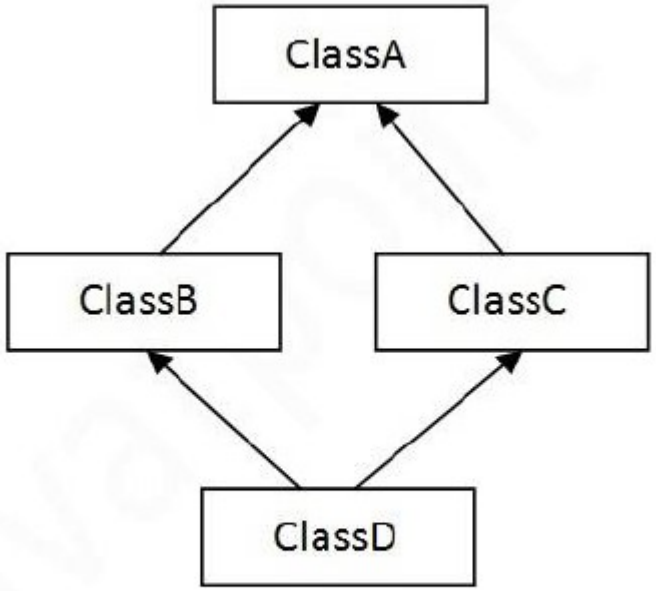
1. Single Level Inheritance
A super class is inherited by only one sub class.
2. Multi Level Inheritance

A sub-class will be inheriting parent class and as well as, the sub-class act as a parent class to other class., and so on.
3. Hierarchical Inheritance

A Super is inherited by many sub classes.



4) Multiple



5) Hybrid

i.e : class C extends A,B

```
{
//CTE Compile Time Error
}
```

4. Multiple Inheritance

A sub class extending more than one super class.

5. Hybrid Inheritance

It is a combination of Multiple and Multilevel inheritance.

Note:

- **Java does not support Multiple Inheritance and Hybrid Inheritance directly with classes.**
- **In Java multiple and hybrid inheritance is supported through interfaces only.**

```
//Program to Demonstrate Single
Level Inheritance -1
class ClassA
{
public void showA()
{
System.out.println("show method of
ClassA");
}
}
class ClassB extends ClassA
{
public void showB()
{
System.out.println("show method of
ClassB");
}
}
public class SingleLevel
{
public static void main(String args[])
{
ClassB b = new ClassB();
b.showA();
b.showB();
}
}
```

```
//Multi Level Inheritance -1
class A{
int i=10;
public void showA()
{
System.out.println(" show() method of
A"+i);
}
}
class B extends A{
int j=25;
public void showB()
{
System.out.println(" show() method of
B"+j);
}
}
class C extends B{
int k=20;
public void showC()
{
System.out.println(" show() method of
C"+k);
System.out.println("The members sum is
" + (i +j +k));
}
}
public class MultiLevel
{
public static void main(String args[])
{
C c = new C();
c.showA();
c.showB();
```

```
//Hierarchical Inheritance -1
class A{
int i=10;
public void showA()
{
System.out.println(" show() method of
A"+i);
}
}
class B extends A{
int j=25;
public void showB() {
System.out.println(" show() method
of B"+j);
}
}
class C extends A{
int j=99;
public void showC() {
System.out.println(" show() method of
C"+j);
}
}
class D extends A{
int j=55;
public void showD() {
System.out.println(" show() method of
D"+j);
}
}
public class HierLevel{
public static void main(String args[]) {

B b = new B();
b.showA();
b.showB();
C c = new C();
```

1. Use super keyword
2. Write programs using StringTokenizer class

1. super keyword

“super” keyword is used in java for

1. Calls the super class constructor.

A subclass can call the constructor defined by its super class in chain.

super (arguments list);

2. To access the member of a super class that has been hidden by the member of a subclass. (Data or method).

If a subclass has the same member as super class, the subclass hides the definition of super class. To access the super class member in subclass uses super keyword.

super.member;

eg: super.i; super.show();

3. super can be used to refer immediate super class instance variable, immediate super class method and immediate super class constructor.

4. super must be the first statement in the sub class constructor.

5. super () is provided by compiler implicitly as default constructor if no constructor is specified explicitly.

//1. Program to access super class constructor

```
class Box {
private double width;
private double height;
private double depth;
Box(Box ob){
width = ob.width;
height = ob.height;
depth = ob.depth;
}
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
Box() {
width = -1;
height = -1;
depth = -1;
}
Box(double len) {
width = height = depth = len;
}
double volume(){
return (width * height * depth);
}
}
class BoxWeight extends Box {
double weight;
BoxWeight(BoxWeight ob) {
super(ob);
weight = ob.weight;
}
```

```
BoxWeight(double w, double h,
double d, double m)
{
super(w, h, d); // call super class
constructor
weight = m;
}
BoxWeight()
{
super();
weight = -1;
}
BoxWeight(double len, double m)
{
super(len);
weight = m;
}
}
```

```
public class SuperDemo1
{
public static void main(String args[])
{
BoxWeight b1 = new BoxWeight(12,23.5);
BoxWeight b2 = new BoxWeight(2, 3, 4,
0.076);
BoxWeight b3 = new BoxWeight();
BoxWeight b4 = new BoxWeight(b2);
System.out.println(b1.volume() + “ :
" +b1.weight);
System.out.println(b2.volume() + " :
" +b2.weight);
System.out.println(b3.volume() + " :
```

```
// 2. Accessing member (data and methods) of super class.
class A {
int i;
A(int x)
{
i=x;
}
void show()
{
System.out.println("A - i is " +i);
}
}
class B extends A{
int i ; //hides super class i
B(int x , int y){
super(x);
i= y;
}
void show() {
super.show(); //calls super class show.
System.out.println("B - i is " +i);
System.out.println(i+2); // subclass i
System.out.println(super.i+19);
}
}
public class SuperDemo2{
public static void main(String[] args) {
B b = new B(12,19);
b.show();
}
}
```

```
// 3 super() is provided by the
compiler implicitly
class Animal
{
Animal()
{
System.out.println("animal is created");
}
}
class Dog extends Animal
{
Dog()
{ // super(); calls automatically.
System.out.println("dog is created");
}
}
```

```
public class SuperDemo3
{
public static void main(String args[])
{
Dog d=new Dog();
}
}
```

Output :
animal is created
dog is Created

/ demonstrate when constructors are called,
The order in which derived is order of constructor execution

```
class A
{ A()
{
System.out.println("Inside A's
constructor.");
}
}
class B extends A
{
B()
{
// super() ; implicitly done by compiler
System.out.println("Inside B's
constructor.");
}
}
class C extends B
{
C()
{
System.out.println("Inside C's
constructor.");
}
}
public class SuperDemo4
{
public static void main(String args[])
{
C c = new C();
}
}
//output: order - A->B->C
```

2. StringTokenizer class

StringTokenizer Class - java.util.*; the processing of text often consists of parsing a formatted input string. Parsing is the division of text into a set of discrete parts or tokens.
The StringTokenizer class provides the first step in this parsing process, often called the lexer (lexical analyzer) or scanner.

The StringTokenizer constructors
StringTokenizer(String str)
StringTokenizer(String str, String delimiters)

StringTokenizer(String str, String delimiters, boolean flag)
The default set of delimiters consists of the whitespace characters: space, tab, newline, and carriage return.
If flag is true, then the delimiter is also returned as token.

The methods define in StringTokenizer are

Method Name	Description
int countTokens( )	The number of tokens left to be parsed and returns the result.
boolean hasMoreElements( )	Returns true if one or more tokens remain in the string
boolean hasMoreTokens( )	Returns true if one or more tokens remain in the string
Object nextElement( )	Returns the next token as an Object.
String nextToken( )	Returns the next token as a String.

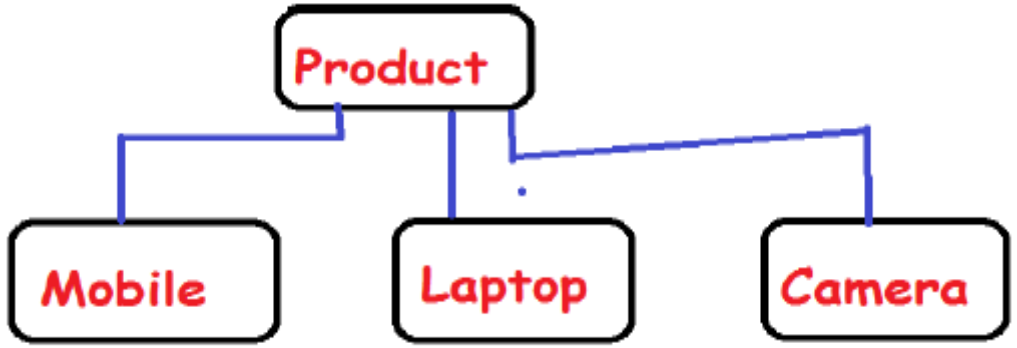
```
//StringTokenizer Exampl-1
import java.util.*;
public class TokenDemo
{
public static void main(String[] args)
{
Scanner sc = new Scanner(System.in);
System.out.println("Enter a line of Text");
String str = sc.nextLine();
StringTokenizer st=new StringTokenizer(str);
System.out.println("The Count of Tokens is " +st.countTokens()); //3
System.out.println("The tokens are");
while(st.hasMoreTokens())
{
String s = st.nextToken();
System.out.println(s);
}
System.out.println("The Count of Tokens is " +st.countTokens()); // 0
System.out.println("Enter a string with # as delimeter");
String str1 = sc.nextLine();
StringTokenizer st1 = new StringTokenizer(str1,"#",true)
System.out.println("The Count of Tokens is " +st1.countTokens()); //5
while(st1.hasMoreTokens())
{
String s = st1.nextToken();
System.out.println(s);
}
}
```

```
//StringTokenizer Exampl-2
import java.util.*;
public class SumDemo
{
public static void main(String[] args)
{
Scanner sc = new Scanner(System.in);
System.out.println("Enter a line of
Integers");
String str = sc.nextLine();
StringTokenizer st=new
StringTokenizer(str);
System.out.println("The Count of
Tokens is " +st.countTokens());
int n,sum=0;
System.out.println("The tokens are");
while(st.hasMoreTokens())
{
String s = st.nextToken();
n=Integer.parseInt(s);
System.out.println(n);
sum=sum+n;
}
System.out.println("Sum of Tokens
is :"+sum);
System.out.println("The Count of
Tokens is " +st.countTokens());// 0
}
}
```

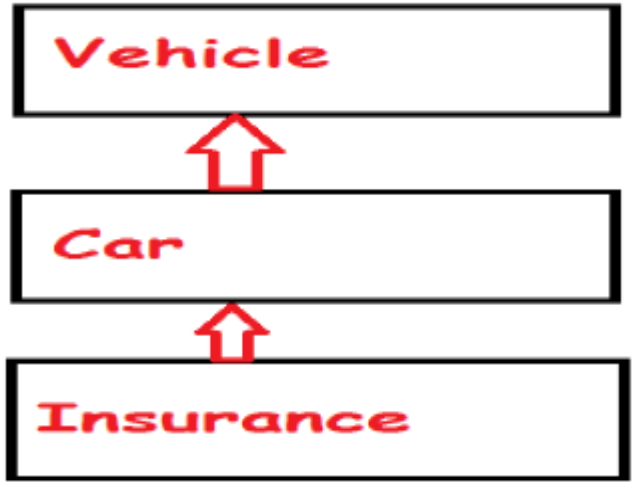
1. Create Objects for real world using Inheritance.

Problem Solving Assignment

1 Create a class hierarchy to demonstrate the following structure with appropriate d



2. Create a Multilevel Structure to demonstrate the following inheritance



Use super and this keyword appropriately.

Member Access and Inheritance

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as private.

```
class A {
int i;
private int j; // private to A
void setij(int x, int y) {
i = x;
j = y;
}
}
class B extends A {
int total;
void sum() {
total = i + j; // Compile time error
}
}
```

```
public class AccessDemo {
public static void main(String
args[])
{
B b1 = new B();
b1.setij(10, 12);
b1.sum();
System.out.println("Total is " + b1.total);
b1.i=20
b1.j=24 //Compilt Time Error
}
}
```

```
//Single level -Inheritance Box example
class Box
{
double width;
double height;
double depth;
Box(){ // Default Constructor
width = -1;
height = -1;
depth = -1;
}
Box(double len) { //Parameterized Constructor
width = height = depth = len;
}
Box(double w, double h, double d) {
width = w; //Parameterized Constructor
height = h;
}
}
Box(Box ob){
width = ob.width; // pass object to constructor-clone of object
height = ob.height;
depth = ob.depth;
}
double volume()
{
return width * height * depth;
}
}
```

```
class BoxWeight extends Box
{
private double weight; // weight of
box
BoxWeight(double w, double h,
double d, double m)
{
width = w;
height = h;
depth = d;
weight = m;
System.out.println("weight is "
+weight);
}
}
```

```
public class InheritDemo3
{
public static void main(String args[])
{
BoxWeight b1 = new BoxWeight(10, 20,
15, 34.3);
double vol;
vol = b1.volume();
System.out.println("Volume of b1 is " +
vol);
// System.out.println("Weight of b1 is " +
b1.weight); //CTE
}
}
```

A Superclass Variable Can Reference a Subclass Object

-A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.

-when a reference to a subclass object is assigned to a superclass reference variable, we will have access only to those parts of the object defined by the superclass.

In the above example

```
public class Demo2 {
public static void main(String
args[])
{
BoxWeight bw = new BoxWeight(3,
5, 7, 8.37);
Box b1 = new Box();
double vol;
vol = bw.volume();
System.out.println("Volume of bw
is " + vol);
System.out.println("Weight of bw is
" + bw.weight);
// assign bw reference to b1
reference
b1 = bw;
vol = b1.volume();
System.out.println("Volume of b1 is
" + vol);
System.out.println("Weight of b1 is
" + b1.weight); //CTE
}
}
```

// Mutilevel Inheritance in Java – 2

Box Example

```
class Box
{
private double width;
private double height;
private double depth;
Box() {
width = -1;
height = -1;
depth = -1;
}
Box(double len) {
width = height = depth = len;
}
Box(double w, double h, double d){
width = w;
height = h;
depth = d;
}
Box(Box ob) {
width = ob.width;
height = ob.height;
depth = ob.depth;
}
double volume()
{
return width * height * depth;
} }
```

```
class BoxWeight extends Box
{
double weight;
BoxWeight()
{
super();
weight = -1;
}
BoxWeight(double len, double m){
super(len);
weight = m;
}
BoxWeight(double w, double h,
double d, double m) {
super(w, h, d);
weight = m;
}
BoxWeight(BoxWeight ob){
super(ob);
weight = ob.weight;
}
}
class Shipment extends BoxWeight
{
double cost;
Shipment() {
super();
cost = -1;
}

Shipment(double len, double m, double
c) {
super(len, m);
cost = c;
}
```

```
Shipment(double w, double h,
double d,double m, double c)
{
super(w, h, d, m); // call superclass
constructor
cost = c;
}
Shipment(Shipment ob){
super(ob);
cost = ob.cost;
}
}
public class MultiLevel2
{
public static void main(String args[])
{
Shipment s1 = new Shipment (10, 20,
15, 10, 564289.236);
double vol;

vol = s1.volume();
System.out.println("Volume of
shipment1 is " + vol);
System.out.println("Weight of
shipment1 is "+ s1.weight);
System.out.println("Shipping cost: " +
s1.cost);
}
}
```

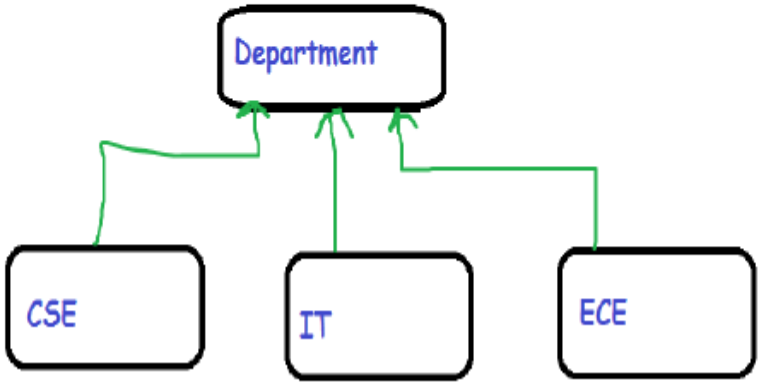
```
//Hierarcical Inheritance 2- Bank
Example
class Bank
{
int acno;
String name;
String type;
double bal;
Bank(int no , String n , String t, double b)
{
acno = no;
name = n;
type=t;
bal =b; }
public void display()
{
System.out.println("The Account Details
are");
System.out.print(acno + " , ");
System.out.print(name + " , ");
System.out.print(type+ " , ");
```

```
class SBI extends Bank
{
double rot; //rate of interest
SBI(int no , String n , String t, double b,
double r){
super( no , n , t, b);
rot = r;
}
public double findTotal()
{
bal = bal + (bal *rot*0.01); //some
Calculation
return(bal);
```

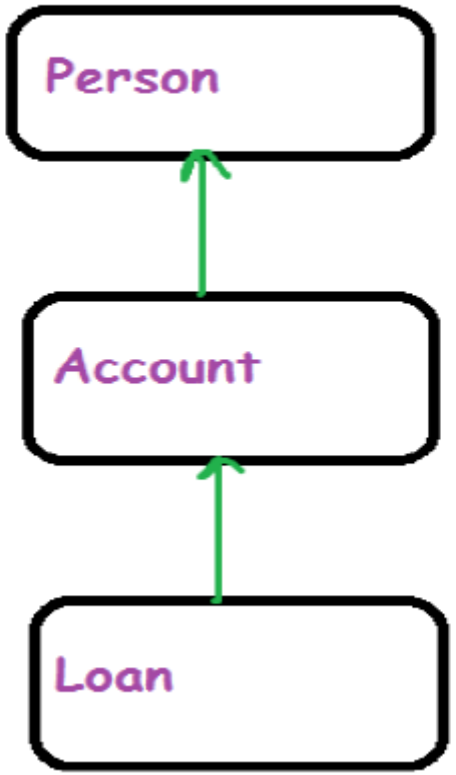
```
class ICICI extends Bank
{
double rot; //rate of interest
ICICI(int no , String n , String t,
double b, double r){
super( no , n , t, b);
rot = r;
}
public double findTotal() {
bal = bal + (bal *rot*0.01);
return(bal);
}
}
class HDFC extends Bank
{
double rot; //rate of interest
HDFC(int no , String n , String t, double b,
double r) {
super( no , n , t, b);
rot = r;
}
public double findTotal() {
bal = bal + (bal *rot*0.01);
return(bal);
}
}
public class HierLevel2{
public static void main(String args[])
{
SBI s = new SBI(12345, "Ganesh",
"Salary",710037.5);
s.display();
System.out.println("Total Amount in SBI is
" +s.findTotal());
HDFC h = new HDFC(12598, "Suresh",
"Salary",123700,8.5);
```

Related Examples of Inheritance

1. Employee – Developer [Developer is a Employee]
2. Hierarchical Structure to Demonstrate



3. Create a Multilevel Structure to demonstrate the following



Use super and this keyword appropriately.

1. Implement Run Time Polymorphism
2. Use final keyword

1. Method Overriding

- "In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its super class, then the method in subclass is said to **override the method** in super class."
- The subclass will hide the super class version.
- Method overriding occurs only when the name and type signature of two methods is identical.
- Method overriding is used to provide specific implementation of a method that is already provided by its super class.
- Note: If super class and sub class has the same method name but different signature (parameter declarations), then the methods are overloaded but not overridden.

Rules for Method Overriding

- Method in sub class must have same name and parameter signature as super class.
- Must be Is-A relationship. (Inheritance).
- We cannot override a main () method because it is a static method belongs to class but not to heap area.
- We can overload a main() method , but Java will call only the main()

```
class A {
int i,j;
A(int a , int b)
{
i=a;
j=b;
}

void show()
{
System.out.println("i , j is " +i + ","
+j);
}
}
class B extends A{
int k;
B(int a , int b , int c) {
super(a,b);
k=c;
}

void show() //hides show() of super
class.
{
// super.show();
System.out.println("k is " +k);
}
}
public class RideDemo{
public static void main(String
args[]) {
B b = new B(7,8,9);
b.show(); //k is 9
}
}
```

2. Runtime Polymorphism [Dynamic Method Dispatch]

- Method Overriding forms the basis for Dynamic method dispatch.
- Method overriding in java provides runtime polymorphism.
- Using DMD, a call to an overridden method is resolved at runtime, rather than at compile time.
- Runtime polymorphism in java implemented by using Dynamic method dispatch.

Upcasting: A super class reference variable can refer to a sub class object. I.e. When reference variable of super class refers to the sub class object, then it is called up casting.

Eg: class A -> A is Super class
class B extends A -> B is sub class

```
A ref;
B b = new B();
ref = b; // upcasting
or
ref = new B(); //upcasting
```

“When an overridden method is called through a super class reference variable, java determines which version of that method to execute based upon the type of object being referred to at the time the call occurs.”

// Dynamic Method Dispatch – Example1

```
class A {
void callme() {
System.out.println("Inside A's callme method");
}
}
class B extends A {
void callme()
{
System.out.println("Inside B's callme method");
}
}
class C extends A {
void callme(){
System.out.println("Inside C's callme method");
}
}
public class DMDDemo {
public static void main(String args[]) {
A a = new A();
B b = new B();
C c = new C();
A r; //reference variable of super class
r = a;
r.callme(); //a version using upcasting
r= b;
r.callme(); // b version - upcasting
r= c;
r.callme(); //c verrsoion - upcasting
r = new B();
//directly assign object to super
```

```
// Using run-time polymorphism-
Example -2
class Figure
{
double dim1;
double dim2;
Figure(double a, double b)
{
dim1 = a;
dim2 = b;
}

double area()
{
System.out.println("Area for Figure
is undefined.");
return 0;
}
}
```

```
class Rectangl extends Figure
{
Rectangl(double a, double b)
{
super(a, b);
}
double area()
{
System.out.println("Inside Area for
Rectangle.");
return dim1 * dim2;
}
}
```

```
class Triangle extends Figure
{
Triangle(double a, double b)
{
super(a, b);
}
double area()
{
System.out.println("Inside Area for
Triangle.");
return (dim1 * dim2) / 2;
}
}
```

```
public class DMDDemo1
{
public static void main(String args[])
{
Figure f = new Figure(10, 10);
Rectangl r = new Rectangl(9, 5);
Triangle t = new Triangle(10, 8);
Figure ref;
ref = r; //Upcasting
System.out.println("Area is " +
ref.area());

ref = t;
System.out.println("Area is " +
ref.area());
ref = f;
System.out.println("Area is " +
ref.area());
```

```
// Dynamic Method Dispatch Example-3
class Bank
{
float getRateOfInterest()
{
return 0;
} }
class SBI extends Bank
{
float getRateOfInterest()
{
return 8.4f;
} }
class ICICI extends Bank
{
float getRateOfInterest()
{
return 7.3f;
} }
class AXIS extends Bank
{
float getRateOfInterest()
{
return 9.7f;
} }
class DMDDemo3
{
public static void main(String args[])
{
Bank b;
b=new SBI();
System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
b=new ICICI();
System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
b=new AXIS();
System.out.println("AXIS Rate of
```

3. final keyword

- The final keyword in java is used to restrict the user.
- final is used with a variable , a method , or a class.

final keyword with a variable

- A data member declared as final prevents its contents from being modified.(Constant)
- I.e. A data variable with final is a keyword.
- We must initialize a final field when it is declared.
- The value can be initialized during declaration or in the constructor.
- A local variable or parameter can also be made as “final”

```
final double PI = 3.1429;
PI = PI + 2; //CTE
```

```
//Example to demonstrate final variable
public class FinalDemo1
{
final int SPEED_LIMIT =90;
void find()
{
SPEED_LIMIT=120;
}
public static void main(String args[])
{
FinalDemo1 d = new FinalDemo1();
d.find(); //Compile time Error
}
}
```

final keyword with a method

- A method declared as final, will not allow overriding.
- i.e. It prevents overriding / stops overriding.
- A constructor cannot be declared as final. Because it is never inherited.
- A final method is inherited, but not overridden.

```
class Bike
{
final void show()
{
System.out.println("Bike's
Method");
}
}
```

```
class Honda extends Bike
{ //inherits show()
void show()
{
System.out.println("Honda's
Show");
}
void display()
{
System.out.println("Honda's
Show");
}
}
public class FinalDEmo2
{
public static void main(String
args[])
{
Honda h1 = new Honda();
```

final keyword with a class

A class declared as final will not allow extending. i.e. It stops inheritance.

```
final class Bike
{
final void show()
{
System.out.println("Bike's Method");
}
}
class Honda extends Bike //Compile time error
{
//trying to inherit - CTE
}
```

final keyword with a Parameter

 If we declare a parameter as final, it cannot allow changing in its method.

```
public class FinalDemo4
{
int find(final int n)
{
n=n+20; //CTE
return (n);
}

public static void main(String args[])
{
FinalDemo4 d = new FinalDemo1();
System.out.println(d.find(10));
//Compile time Error
}
}
```

A final variable which is not initialized at the time of declaration is called “blank final variable”. A blank final variable must be initialized in constructor only. Once it is initialized, cannot be modified.

```
class Person
{
final String PAN; //not initialized.
Person( )
{
PAN = "AAIPA2346G";
System.out.println(PAN);
} }
public class FinalDemo5
{
public static void main(String args[])
{
Person p = new Person();
```

A static blank final variable must be initialized in static block.

```
public class FinalDemo5
{
static final String PAN; //not initialized.
static{
PAN = "AAIPA2346G";
System.out.println(PAN);
}
public static void main(String args[])
{
Person p = new Person();
}
}
```

1. Implement Abstraction in OOP

Problem Solving Assignment

1. Program to demonstrate Multiple Inheritance using Interfaces
2. Compare Classes and Abstract Classes
3. Program to create Shared variables using interface and use in classes.

1. Abstract classes

- Abstraction is a process of hiding implementation details and showing only functionality to the users.
  - Eg: Sending a Message, Driving a car etc.
- Abstraction in java achieved using
  - Abstract classes (0 to 100%)
  - Interfaces (100%)

Abstract classes

Creating a super class that only defines a generalized form that will be shared by all of its sub classes; the sub classes must provide implementation.

The method which is implements by the sub class must be specified as **“abstract”** type modifier. These methods are referred as **“sub classer responsibility.”**

Any class that contains one or more abstract methods, then the class is defines as abstract class.

-An abstract method has no body defined in it.

-An abstract class is non-concrete class.

Concrete class is a class that has an implementation for all of its methods. They cannot have any unimplemented methods. It can also extend an abstract class or implement an interface as long as it implements all their methods. It is a complete class and can be instantiated.

Abstract type is a class that has no implementation for some of the methods defined in it. It is an incomplete class and cannot be instantiated.

```
abstract double area(double d1 ,
double d2);
abstract void show();
abstract class Figure
{
double dim1,dim2;
Figure(double d1 , double d2)
{
dim1 = d1;
```

```
abstract double area( ); //abstract
method

void display() //non abstract method
– concrete method
{
System.out.println(dim1);
System.out.println(dim2);
```

Rules for abstract modifier

- An abstract method has no method body
- A class which has at least one abstract method then the class must be declared a s abstract.
- All abstract methods of super class must be implemented in subclass by overriding.
- If an abstract method is not implemented in sub class, then the sub class is also made as abstract.
- No objects of an abstract class exist. i.e. An abstract cannot be directly instantiated with the new operator.
- An abstract class not fully defined. Hence object cannot be created.
- Can not declare abstract constructors or abstract static methods.
- Abstract classes can be used to create object references because of Dynamic Method Dispatch and Run time polymorphism

```
//Example 1: To demonstrate
abstract class.
abstract class A
{
abstract void show ();
public void display()
{
System.out.println("This is non
abstract method");
}
}
class B extends A
{
void show() //implements abstract
method
{
System.out.println("Abstract
method Implemented");
}
}
```

```
public class AbstractDemo1
{
public static void main(String args[])
{
// A a = new A(); // Compile time
error
B b = new B();
b.show();
b.display();
}
}
```

```
//Example 2 : To demonstrate
abstract class.
abstract class Figure
{
double dim1;
double dim2;
Figure(double a, double b)
{
dim1 = a;
dim2 = b;
}
abstract double area(); //abstract
method
public void display()
{
System.out.println("The dimensions
are");
System.out.println(dim1);
System.out.println(dim2);
}
}
class Rectangl extends Figure
{
Rectangl(double a, double b)
{
super(a, b); }
double area()
{
System.out.println("Inside Area for
Rectangle.");
return dim1 * dim2;
} }
class Triangle extends Figure
{
Triangle(double a, double b)
{
super(a, b); }
double area()
{
```

```
public class AbstractDemo2
{
public static void main(String args[])
{
Figure f = new Figure(10, 10); //CTE
Rectangl r = new Rectangl(9, 5);
Triangle t = new Triangle(10, 8);
Figure ref;
ref = r; //upcasting
ref.display();
System.out.println("Area of Rectangle is " + ref.area());
ref = t;
ref.display();
System.out.println("Area of Triangle is " + ref.area());
}
}
```

```
//Example 3: To demonstrate
abstract class.
abstract class Bank{
abstract int getRateOfInterest();
}
class SBI extends Bank
{
int getRateOfInterest()
{
return 7;
}
}
class PNB extends Bank
{
int getRateOfInterest()
{
return 8;
}
}
```

```
class AbstractDemo3
{
public static void main(String args[])
{
Bank b;
b=new SBI();
System.out.println("Rate of Interest
is: "+b.getRateOfInterest());
b=new PNB();
System.out.println("RateofInterest
is: "+b.getRateOfInterest());
}
}
```

2. Interfaces

- **An Interface allows creating a fully abstract class.**
- **An interface specifies what a class must do but not how it does it.**
- **An Interface has**
  - **Methods declared without any body**
  - **Final and static data members. (No Instance variables).**
- **Once an interface is defined, any number of classes can implement an interface.**
- **The class which implements an interface must provide definition for all the methods of an interface.**
- **The keyword interface allows defining an interface.**
- **An interface in java is a blueprint of a class.**
- **The interface in java is** a mechanism to achieve abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve abstraction and multiple inheritance in Java.**
- **Java Interface also represents IS-A relationship.**
- **It cannot be instantiated just like abstract class.**

Defining an Interface

```
access interface InterfaceName
{
return_type
methodName1(parameters list);
return_type
methodName2(parameters list);
-----------------------------
return_type
methodNamen(parameters list);
type final_val_1 = value;
type final_val_2 = value;
-----------------------------
type final_val_n = value;
}
```

- All methods and variables are implicitly public and abstract.
- All variables are implicitly final and static, must be initialized.

Eg: interface IntDemo

```
{
void callme(int x);
}
```

Implementing Interfaces

To implement an interface, use "implements" clause in the class definition and provide definition for all the methods defined by the interface.

```
access ClassName implements
interface1 , interface2
{
----------------
Class body //implementation of
methods
```

The type signature of implementing class must match with the interface defined methods.

Eg: class Demo implements IntDemo

```
{
public void callme(int x)
{
System.out.println(“The Interface method implemented”);
System.out.println(“The value is “+x);
}
}
```

Accessing Implementation through Interface references

```
class Test
{
public static void main(String args[])
{
IntDemo i = new Demo();
i.callme();
/* using class instance
Demo d = new Demo();
d.callme(); */
}
}
```

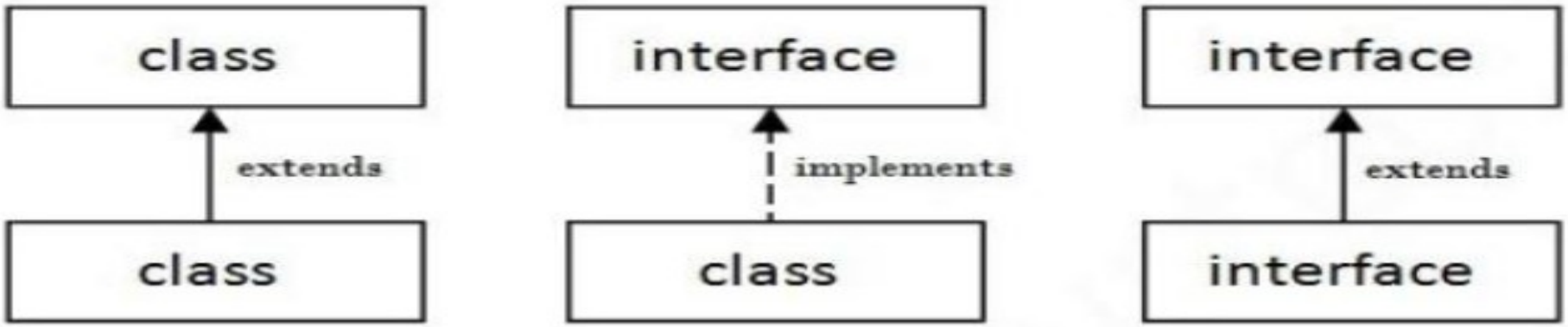
```
// Example to demonstrate an
interface.
interface Bank
{
float rateOfInterest();
}
class SBI implements Bank
{
public float rateOfInterest()
{
return 9.15f;
}
}
class PNB implements Bank
{
public float rateOfInterest()
{
return 9.7f;
}
}
class IntDemo2
{
public static void main(String[]
args)
{
Bank b=new SBI();
System.out.println("ROI:
"+b.rateOfInterest());
}
}
```

Variable in Interfaces

- By default the variables declared in interface are public static final.
- Variable are used in an interface to import Shared Constants into multiple classes.
- All variables of an interface must be initialized before to use.

```
//interfaces with shared variables
interface SharedData
{
int ADD=1;
int SUB=2;
int MUL=3;
double PI=3.142;
double SQRT2=1.414; }
class IntDemo3 implements
SharedData
{
static void perform( int ch)
{ int a=20,b=5,c;
{
case 1 : c=a+b;
System.out.println("add " +c);
break;
case 2 : c=a-b;
System.out.println("sub " +c);
break;
case 3 : c=a*b;
System.out.println("mul " +c);
break;
} }
public static void main(String[] args)
{
perform(ADD);
perform(SUB);
perform(MUL);
System.out.println("The SQRT of 2 is
" +SQRT2);
System.out.println("The area of a
circle is "+(PI * 4 *4));
} }
```

Relationship between class and Interface



Interfaces can be extended

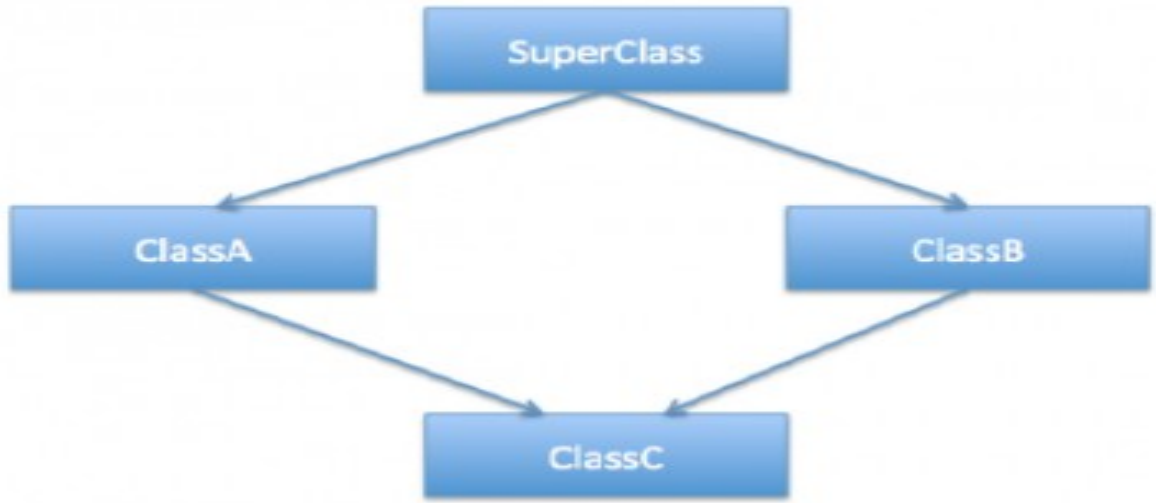
- **An interface can extend one or more other interfaces. Thus provides interface inheritance.**
- **A class which implements an interface must provide definition of all the methods of inheritance chain.**
- **The inheritance with interfaces can be**

1. Single level inheritance
2. Multi level inheritance
3. Multiple inheritance
4. Hierarchical inheritance
5. Hybrid Inheritance.

Multiple inheritance is not supported through class in java, but it is possible by an interface.

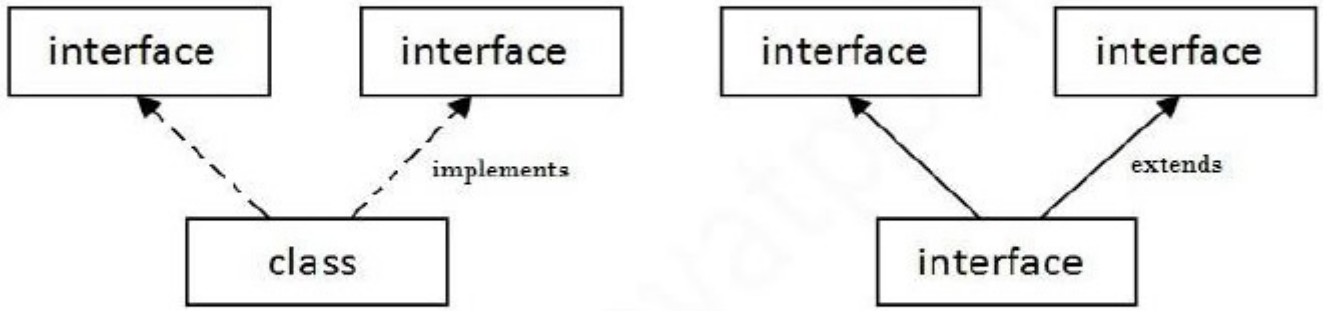
Diamond Problem in Java

To understand diamond problem easily, **let's assume that multiple inheritances were supported in java**. In that case, we could have a class hierarchy like below image.



- **Multiple inheritance is not supported in the case of class because of ambiguity, it is called Diamond Problem.**
- **Hence for simplicity and resolve ambiguity java does not support multiple inheritance using classes.**

Multiple Inheritance in Java by interface



Multiple Inheritance in Java

```
Eg: class SuperClass
{
void print()
{
System.out.println("Inside Super class");
}
}
// First Parent class
class A extends SuperClass
{
void print()
{
System.out.println("Inside Class A");
}
}
```

```
// Second Parent Class
class B extends SuperClass
{
void print()
{
System.out.println("Inside Class B");
}
}
// Error : Test is inheriting from
multiple classes
```

```
class Test extends A, B
{
public static void main(String args[])
{
Test t = new Test();
t.print(); //CTE - ambiguity
}
}
```

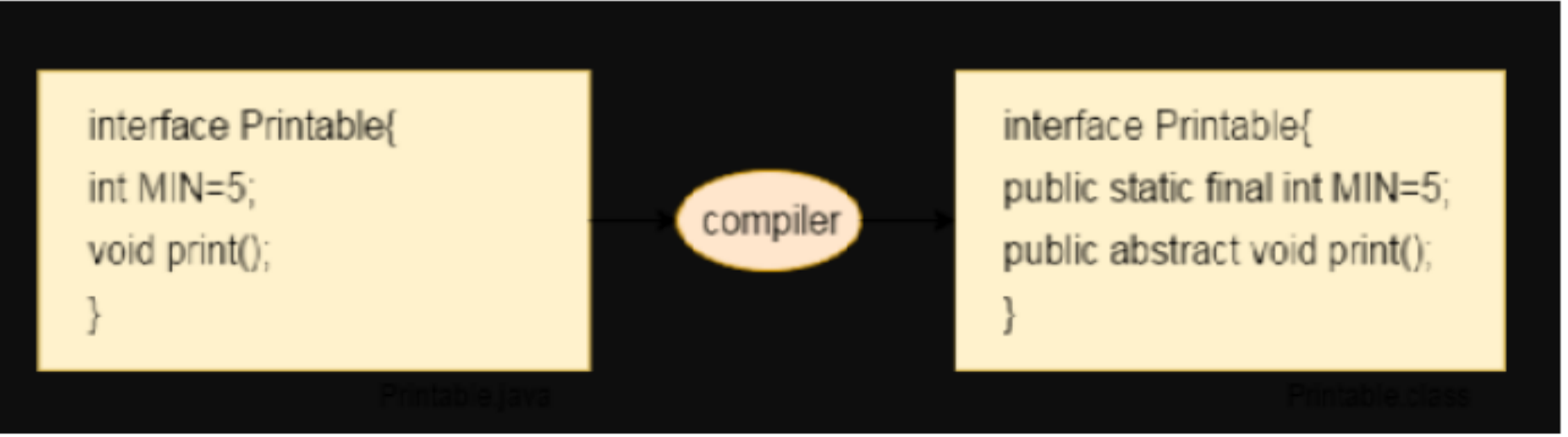
However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class.

```
interface A
{
void print();
}
interface B
{
void print();
}
class MultiTest implements A,B
{
public void print()
{
System.out.println("Allows Multiple
inheritance");
}
public static void main(String args[])
{
MultiTest obj = new MultiTest();
obj.print();
}
}
```

Key points of Interfaces is

- Blue Print of a class
- Provides complete abstraction. (100%)
- Also represents Is-A relationship.
- Cannot be instantiated. (Object cannot be created)
- ✓ Used to achieve multiple inheritance.
- By default all methods are public and abstract.
- By default all data members are static, final.

Eg: int a=10; is
public static final int a=10;



- ✓ Implements keyword is used by classes to implement an interface
- ✓ When a class implements any interface method, it must be declared as public.
- ✓ A class must implement all the methods of an interface otherwise, the class must be declared as abstract.
- ✓ Interface cannot be declared as private or protected.
- ✓ Interface variables must be initialized at the time of declaration otherwise compile time error.
- ✓ A class can implement any number of interfaces. Hence provides multiple inheritance.

```
interface A {
}
Interface B {
}
class C implements A,B
{
//Implementation of A and B
}
```

- If there are two or more same methods in two interfaces and a class implements both interfaces; Implementation is required only for one method.
- A class cannot implement two interfaces that have methods with same name but different return type.
- Variable name conflicts can be resolved by interface name.

```
interface A
{
int X=10;
}
Interface B
{
int X=20;
}
```

```
class C implements A,B
{
public static void main(String
args[])
{
System.out.println(X);
System.out.println(A.X);
System.out.println(B.X);
}
}
```

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

Abstract class	Interface
1) Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods. Since Java 8, it can have default and static methods also.
2) Abstract class doesn't support multiple inheritance.	Interface supports multiple inheritance.
3) Abstract class can have final, non-final, static and non-static variables.	Interface has only static and final variables.
4) Abstract class can provide the implementation of interface.	Interface can't provide the implementation of abstract class.
5) The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.

6) Example:
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

ace that

Simply, abstract class achieves partial abstracti on (0 to 100%) whereas interface achieves fully abstracti on (100%).

```
//Example Program on Interfaces -1
interface Bank
{
float rateOfInterest();
}
class SBI implements Bank
{
public float rateOfInterest()
{
return 9.15f;
}
}
class PNB implements Bank
{
public float rateOfInterest()
{
return 9.7f;
}
}
```

```
class IntDemo2
{
public static void main(String[]
args)
{
Bank b1=new SBI(); //upcasting
System.out.println("ROI: of SBI
"+b1.rateOfInterest());
Bank b2=new PNB();
System.out.println("ROI: of PNB
"+b2.rateOfInterest());
}
}
```

```
//Example Program on Interfaces -2
Methods and Interfaces
/* abstract class Employee {
final String company;
final String address;
final int pin;
abstract void show();
} */
interface Employee { //defining
interface
String company="Infosys";
String address="hyd";
int pin= 501232;
void show();
}
class Emp implements Employee {
//implementing interface
public void show() {
System.out.println(company);
System.out.println(address);
System.out.println(pin);
}
}
public class IntDemo4 { //accessing
interface
public static void main(String args[])
{
Emp e1 = new Emp();
e1.show();
}
}
```

Extending interfaces

```
//Example Program on Interfaces -3
interface A{
void showA();
}
interface B extends A{
void showB();
}
interface C extends B{
void showC();
}
class D implements C {
//implementation class
public void showA() {
System.out.println("I am from
ShowA");
}
public void showB(){
System.out.println("I am from
ShowB");
}
public void showC() {
System.out.println("I am from
ShowC");
}
}
public class MLevel{
public static void main(String args[])
{
D d = new D();
```

/ A real-world example:

Let’s consider the example of vehicles like bicycle, car, bike……., they have common functionalities. So we make an interface and put all these common functionalities. And lets Bicycle, Bike, car ….etc implement all these functionalities in their own class in their own way.

```
interface Vehicle {
// all are the abstract methods.
void changeGear(int a);
void speedUp(int a);
void applyBrakes(int a);
}
```

```
class Bicycle implements Vehicle{
int speed;
int gear;
public void changeGear(int
newGear){
gear = newGear;
}
public void speedUp(int increment){
speed = speed + increment;
}
public void applyBrakes(int
decrement){
speed = speed - decrement;
}

public void printStates() {
```

```
class Bike implements Vehicle {
int speed;
int gear;
public void changeGear(int
newGear){
gear = newGear;
}
public void speedUp(int increment){
speed = speed + increment;
}
public void applyBrakes(int
decrement){
speed = speed - decrement;
}
public void printStates() {
System.out.println("speed: " +
speed
+ " gear: " + gear);
}
}
```

```
Public class IntDemo {
public static void main (String[]
args) {
Bicycle b1 = new Bicycle();
b1.changeGear(2);
b1.speedUp(3);
b1.applyBrakes(1);
System.out.println("Bicycle present
state :");
b1.printStates();
Bike bike = new Bike();
bike.changeGear(1);
bike.speedUp(4);
bike.applyBrakes(3);
System.out.println("Bike present
```

New features of Interfaces

1. We can add default implementation for interface methods. This default implementation has special use and does not affect the intention behind interfaces. Suppose we need to add a new function in an existing interface. Obviously the old code will not work as the classes have not implemented those new functions. So with the help of default implementation, we will give a default body for the newly added functions. Then the old codes will still work.

```
interface In1
{
final int a = 10;
default void display()
{
```

```
class TestClass implements In1
{
// Driver Code
public static void main (String[]
args)
{
TestClass t = new TestClass();
t.display(); //hello
}
}
```

2. We can define static methods in interfaces which can be called independently without an object. Note: these methods are not inherited.

```
interface Interface1
{
final int a = 10;
static void display()
{
System.out.println("hello");
}
}
// A class that implements the interface.
class TestClass implements Interface1
{
// Driver Code
public static void main (String[] args)
{
Interface1.display();
}
}
```

3. Interface allows private methods, static methods and private static